DSA Tutorial Week 4 - Sorting

Problem 0 - Preparation *!examine before the tutorial!*

Illustrate the execution of **merge-sort** ,**heap-sort** and **quicksort** on an input array containing 1,7,4,6,2,8,3,5. For the quicksort illustration, select the middle element as pivot.

Use a diagram of a tree to clearly visualize the steps of the algorithms. Check your answers on the website https://visualgo.net/en/sorting. By running the input through each of the algorithms on the interactive website.

A. Draw a tree diagram to illustrate the execution of **merge-sort, heap-sort and quick-sort**. You can check your answers using https://visualgo.net/en/sorting. Use the following int array as input: [17,4,6,2,8,3,5]

B. Take another look at the algorithm that you optimized in week 1.
   Here is an algorithm that for any given array calculates the sum of its unique elements:

```
// Finds the sum of all unique elements in an array
public static int sumOfUniqueElements(int[] array) {
    int sum = 0;
    for (int i = 0; i < array.length; i++) {
        boolean isUnique = true;
        for (int j = 0; j < array.length; j++) {
            if (i != j && array[i] == array[j]) {
                isUnique = false;
                break;
            }
        }
        if (isUnique) {
            sum += array[i];
        }
    }
    return sum;
}
```

Input: An array of integers (array), e.g.: {-1, 2, -1, 2, 3}
Output: The sum of all unique elements in the array, e.g.: 4 (-1+2+3)
Process: The algorithm iterates through each element of the array (i). For each element, it checks whether this element appears elsewhere in the array (j). If a duplicate is found, the element is not considered unique. If no duplicates are found, the element is added to the sum.

Given what you've learnt this week about sorting, can you apply a new optimization to this algorithm? In that case, what will its worst-case complexity be?

EXAMPLE SOLUTION

Optimized Algorithm Using Sorting

- First, sort the array. This can be done using an efficient sorting algorithm like merge sort, quicksort, or even built-in sorting functions, most of which offer O(n log n) complexity.
- After sorting, iterate through the array. Compare each element with its neighbors to determine if it's unique. Since the array is sorted, **a unique element will not be equal to its immediate neighbors.**
- If an element is identified as unique, add it to the sum.

```
public static int sumOfUniqueElementsOptimized(int[] array) {
    Arrays.sort(array); // Sorts the array, O(n log n)
    int sum = 0;

    for (int i = 0; i < array.length; i++) {
        // Check if the current element is unique
        if ((i == 0 || array[i] != array[i - 1]) &&
                (i == array.length - 1 || array[i] != array[i + 1])) {
            sum += array[i];
        }
    }
    return sum;
}
```

Complexity Analysis

1. **Sorting**: The initial sorting of the array is **O(n log n)**.
2. **Iteration**: The single pass through the sorted array is **O(n)**.

Thus, the worst-case complexity of the optimized algorithm is dominated by the sorting step, making it **O(n log n)**, which is an improvement over the original algorithm's **O(n²)** complexity due to the nested loops.

Problem 1 - Heaps

Consider the array `A = [29, 18, 10, 15, 20, 9, 5, 13, 2, 4, 15]`.

In order to solve the problems below, it may help to consider drawing this array as a max-heap in the form of a binary tree. For any given node, the left child is located at index ***2 i + 1***. The right child is located at index ***2i + 2***, where ***i*** is the index of the node in the array.

a. Does **`A`** satisfy the *max-heap* property? If not, fix it by swapping two elements.

b. Using array **`A`** (possibly corrected), illustrate the execution of the heap-extract-max algorithm, which extracts the max element and then rearranges the array to satisfy the max-heap property. For each iteration or recursion of the algorithm, write the content of the array **`A`**.

TEACHER POINTERS

- Encourage students to **draw the given array as a binary tree**. This visual representation helps in understanding how a max-heap is structured and how it functions.
- **Emphasize the definition of a max-heap:** In a max-heap, every parent node is greater than or equal to its child nodes. Guide students to check each node in the array against this property, starting from the root. Point out that checking should be done for all non-leaf nodes.
- **Fixing the Heap:** Discuss how to correct the heap if the max-heap property is violated. Usually, this involves swapping elements to restore the property. Highlight the importance of **choosing the right element to swap** to ensure the heap is correctly restructured.
- Clarify how max-heapify works – it ensures that the subtree with the replaced root satisfies the max-heap property.
- **Algorithm Iteration:** Encourage students to follow each iteration of re-heapifying until the array satisfies the max-heap property at all levels.
- **Final State of Array:** Ask students to write down the final state of the array after performing heap-extract-max and to verify if the array still maintains the max-heap property.
- Highlight that the same data structure (in this case, a heap) **can be represented in different ways** in memory, such as an array or a tree. Discuss the pros and cons of each representation.

EXAMPLE SOLUTION

Checking Max-Heap Property:
A max-heap is a complete binary tree where the value of each node is greater than or equal to the values of its children. We check this property for each non-leaf node in the array A.

- Node 29 (Index 0): Children are 18 and 10. Both are less than 29. ✓
- Node 18 (Index 1): Children are 15 and 20. 20 is greater than 18. ✗
- Node 10 (Index 2): Children are 9 and 5. Both are less than 10. ✓

Since node 18 has a child greater than itself, the array A does not satisfy the max-heap property.

Fixing the Heap: Swap the parent node 18 with its child node 20 to correct the heap.
`A=[29,20,10,15,18,9,5,13,2,4,15]`

Heap-Extract-Max Algorithm Execution

Starting Array (Corrected Array from Problem 1): `A=[29,20,10,15,18,9,5,13,2,4,15]`

Extract Max (the root of the heap, Index 0):
- Max element: 29
- Swap 29 with the last element in the heap (15, Index 10).
- Remove the last element (now 29).
- Resulting Array: 15,20,10,15,18,9,5,13,2,4 (29 extracted)

Max-Heapify (Re-heapify starting from the root, Index 0):
- Compare new root (15) with its children (20 and 10).
- 20 is the largest. Swap 15 with 20.
- Resulting Array: 20,15,10,15,18,9,5,13,2,4

Max-Heapify (Continue from the swapped child, Index 1):
- Compare node 15 (Index 1) with its children (15 and 18).
- 18 is the largest. Swap 15 with 18.
- Resulting Array: 20,18,10,15,15,9,5,13,2,4

Max-Heapify (Continue from the swapped child, Index 4):
- Node 15 (Index 4) does not have children larger than itself.
- No further swaps needed.
- Final Array: 20,18,10,15,15,9,5,13,2,4

Final State of the Array After Heap-Extract-Max:

```
A=[20,18,10,15,15,9,5,13,2,4]
```

The max element (29) has been extracted, and the array has been rearranged to maintain the max-heap property. Each step of the heapify process ensures that the largest element among the node and its children moves up, maintaining the heap's structure.

Problem 2 - Pancake Flipping

You are in front of a stack of pancakes of different diameters. Unfortunately, you cannot eat them unless they are sorted according to their size, with the biggest one at the bottom. To sort them, you are given a **spatula** that you can use to split the stack in two parts and then flip the top part of the stack. Write the pseudocode of a function **`sortPancakes`** that sorts the stack.

The *i-th* element of array pancakes contains the diameter of the *i-th* pancake, counting from the bottom. The sortPancakes algorithm can modify the stack only through the **`spatulaFlip`** function whose interface is specified below.

```
// Flips over the stack of pancakes from position pos and returns the result
int[] spatulaFlip(int pos, int[] pancakes);

// Returns all pancakes in sorted order
int[] sortPancakes(int[] pancakes) {
```

1. ***After writing the pseudocode***, implement both the `spatulaFlip` and `sortPancakes` function in a programming language of your choice.
2. What algorithm that you know does the flipping sort resemble?

3. An example implementation can be found in the “answers” directory. Though try to implement it yourself first.
4. Test your implementation with various **arrays of pancake sizes** to ensure it correctly sorts them.

Pointer: Notice that you can move a pancake at **position x** to **position y**, without modifying the positions of the order of the other pancakes, using *a sequence of spatula flips*.

TEACHER POINTERS

- **Introducing the Problem:** Start by explaining the problem in a relatable way. Maybe use real-world analogies like the stack of pancakes to illustrate the sorting challenge. Or use the whiteboard to illustrate a small example.
- **Discussing Pseudocode:** Encourage students to first write pseudocode for the sortPancakes function. Focus on the logic and steps involved in sorting the pancakes using flips.
- **Understanding Spatula Flips:** Explain the `spatulaFlip` function and how it's used to change the order of pancakes. Make sure students understand how a flip at position pos affects the stack.
- **Algorithmic Strategy:** Guide students to see how they can apply a selection sort-like strategy: repeatedly moving the largest unsorted pancake to its correct position.
- **Flipping Strategy:** Discuss the strategy for flipping pancakes: bringing the largest pancake to the top and then flipping it to the bottom of the sorted part.
- The sortPancakes solution described in the example represents a variation of the Selection Sort algorithm.

Example Stack of Pancakes:

Let's say the pancakes are in the following order from top to bottom: `[3, 1, 4, 2]`. The goal is to sort them so that the largest is at the bottom and the smallest is on top.

Step-by-Step Example of the `sortPancakes` Algorithm:

1. **Initial Stack**: `[3, 1, 4, 2]`
2. **Find the Largest Pancake**:
   - In this case, the largest pancake (4) is in the 3rd position from the top.
3. **First Flip**:
   - Flip at position 2 (0-based index) to bring pancake 4 to the top.
   - **Stack After Flip**: `[4, 1, 3, 2]`
4. **Second Flip**:
   - Flip the entire stack to move pancake 4 to the bottom.
   - **Stack After Flip**: `[2, 3, 1, 4]`
5. **Find the Next Largest Pancake**:
   - The next largest pancake is 3, which is now at the second position from the top.
6. **Third Flip**:
   - Flip at position 1 to bring pancake 3 to the top.
   - **Stack After Flip**: `[3, 2, 1, 4]`
7. **Fourth Flip**:
   - Flip the top three pancakes to move pancake 3 to the bottom of the unsorted part.
   - **Stack After Flip**: `[1, 2, 3, 4]`

At this point, the stack is sorted in ascending order from top to bottom: `[1, 2, 3, 4]`. The largest pancake is at the bottom, and the smallest is on top, as required.

EXAMPLE SOLUTION

```
function sortPancakes(pancakes):
    n = length of pancakes

    for size = n down to 2:
          // Find the index of the largest pancake in the unsorted part
          maxIndex = 0
          for i = 1 to size:
              if pancakes[i] > pancakes[maxIndex]:
                  maxIndex = i

            if maxIndex != size - 1:
            // Flip the largest pancake to the top if it's not already there
            if maxIndex != 0:
                  pancakes = spatulaFlip(maxIndex, pancakes)

             // Flip the top part to move the largest pancake to the bottom
             pancakes = spatulaFlip(size - 1, pancakes)

return pancakes
```

The `sortPancakes` solution described in the example represents a **variation of the Selection Sort** algorithm.

Explanation:

1. **Outer Loop (size)**: Iterate from the size of the stack down to 2. This loop progressively reduces the portion of the stack that needs sorting, similar to selection sort.
2. **Inner Loop (maxIndex)**: Find the largest pancake in the unsorted portion of the stack. This is the pancake that needs to be moved to its correct position at the bottom of the sorted portion.
3. **First Flip (if maxIndex != 0)**: If the largest pancake is not already at the top, flip the stack at \`maxIndex\` to bring the largest pancake to the top.
4. **Second Flip (size - 1)**: Flip the entire unsorted portion of the stack to move the largest pancake from the top to the bottom of the unsorted portion, effectively placing it in its final position.

Problem 3 - Divide and Conquer

Given an array **A** and a positive integer **k**, the selection problem amounts to finding the largest element **x** ∊ **A** such that at most **k-1** elements of **A** are less than or equal to **x**, or nil if no such element exists. A simple way to implement it is as follows:

```
SimpleSelection(A, k)
      if k > A.length
            return nil
```

```
    else sort A in ascending order
          return A[k-1]
```

Write *another* algorithm named **QuickSelect** that solves the selection problem ***without first sorting*** **A**. Use a divide-and-conquer strategy that "divides" **A** using one of its elements.

Use a divide-and-conquer approach similar to QuickSort, but instead of sorting the entire array, recursively partition the array to find the k-th smallest element.

Also, illustrate the execution of the algorithm on the following input by writing its state at each main iteration or recursion.

```
A = [29, 28, 35, 20, 9, 33, 8, 9, 11, 6, 21, 28, 18, 36, 1]
k = 6
```

After writing the pseudocode for **QuickSelect**, implement it in Java.

Reflection

- Does QuickSelect have a different worst-case time-complexity than QuickSort? How about the average case or expected case?
- Which algorithm will perform the task quicker do you think and why?

TEACHER POINTERS

- It may help students to go through the example first, before trying to come up with the algorithm. Perhaps you can work it out on the board for them so they understand the steps involved and how they differ from what they know about QuickSort.
- Discuss how **divide-and-conquer** breaks down a problem into smaller subproblems, solves each subproblem, and combines their solutions. Students may struggle with the idea that they have to "break-up" a sorting algorithm to adjust it, so guide them through this idea.
- Highlight the similarities and differences between QuickSelect and QuickSort. Both use partitioning, but *QuickSelect* only recurses into **one part of the array**. This also gives way to the discussion regarding the **complexity**.
- Though choosing a **pivot** is crucial, it is not an important part of the learning experience here. A random selection will suffice (as it does in most cases).
- Explain the partition process and how it rearranges elements around the pivot.
- Emphasize the recursive nature of the algorithm and how each recursive call **narrows down** the search space. Again, this is helpful in figuring out the complexity.
- To find the complexity, draw the **recursion tree** of QuickSelect and reason about what happens at each level of the tree.

QuickSelect Execution:

1. **Initial Call**: `QuickSelect(A, 3)`
2. **First Partition (Using 9 as the pivot)**:
    - Rearrange `A` so that elements less than 9 come before it, and those greater come after.
    - After partitioning, `A` might look like this: `[7, 3, 5, 4, 9, 12, 19]`.
    - The pivot element (9) is now in position 4 (0-based index).
3. **Check Pivot Position**:
    - We are looking for the element at position `k = 3` (1-based index), which corresponds to index 2 (0-based index) in the array.

    - Since the pivot (9) is at position 4, we need to focus on the left partition (elements before the pivot).
4. **Recursive Call on Left Partition**:
    - We now call QuickSelect on the left partition `[7, 3, 5, 4]` with the same `k = 3`.
5. **Second Partition (Using 4 as the pivot)**:
    - Partition `[7, 3, 5, 4]` around 4.
    - After partitioning, the left partition might look like this: `[3, 4, 7, 5]`.
    - The pivot element (4) is now in position 1.
6. **Check Pivot Position in Left Partition**:
    - We still need the element at position 2 (0-based index).
    - Since the pivot (4) is at position 1, we focus on the right side of the pivot in the left partition.
7. **Final Step**:
    - The remaining elements to consider are `[7, 5]`.
    - The third smallest element is the smallest element in this subarray.
    - Sorting or simply comparing the two elements, we find that 5 is the third smallest element.

The third smallest element in `A` is 5.

Explanation of the Worst-Case Time Complexity:

- **QuickSort:** In QuickSort, the worst-case occurs when the pivot chosen ends up being the smallest or largest element in the array. This leads to highly unbalanced partitions, where one partition contains all the elements except the pivot.
  In such cases, QuickSort degrades to a quadratic complexity because it has to perform partitioning for each element, leading to $O(n^2)$ time complexity.
- **QuickSelect:** Similar to QuickSort, the worst-case for QuickSelect occurs when the pivot chosen ends up being the smallest or largest element, leading to unbalanced partitions.
  Since QuickSelect involves partitioning and then recursing only on one part of the array, it might seem like it should be faster. However, in the worst case, it still needs to partition the entire array for each recursive call, which also leads to $O(n^2)$ time complexity. The explanation is the same as the QuickSort worst case, at each level, select the worst possible pivot.

Considerations:

- **Average Case vs. Worst Case:** It's important to note that while their worst-case complexities are the same, their average-case complexities differ. QuickSort's average-case complexity is $O(n \log n)$, which it achieves with good pivot choices. QuickSelect, used for selection tasks, has an average-case complexity of $O(n)$, making it faster than QuickSort for its specific purpose in typical scenarios.
- **Pivot Selection:** The pivot selection method greatly influences the actual performance of both algorithms. Strategies like choosing a random pivot or the median-of-three can help avoid the worst-case scenario in most practical cases.

Breakdown of the *Average-Case* Complexity for QuickSelect:

- Like QuickSort, QuickSelect initially **partitions** the array around a pivot. However, unlike QuickSort, it only needs to recurse into one part of the array – the part that contains the k-th smallest element. This is the key difference that affects the complexity.

- On average, the pivot tends to split the array into two roughly equal parts. This doesn't always happen, but over many runs and with a good pivot selection strategy (like choosing a random pivot), it's a reasonable expectation.
- After partitioning, **QuickSelect only processes one half of the array**. Let's assume that on average, each recursive call processes a half of the current array size.
- Cumulative Work:
    - In the first call, QuickSelect processes the entire array of size n.
    - In the next call, it processes an array of size *n/2.*
    - This continues with sizes *n/4, n/8,* and so on.
- The total work done across all these calls forms a geometric series: ***n + n/2 + n/4 + n/8 + ...***
- The sum of this infinite series converges to **2n**.
- Since the sum of work done is linearly proportional to **n**, the average-case complexity of QuickSelect is ***O(n)***.

EXAMPLE SOLUTION

To solve the selection problem using a divide-and-conquer approach without sorting the entire array, we can use an algorithm similar to the QuickSelect algorithm, which is based on the partition method used in QuickSort. The idea is to partition the array around a pivot element such that elements less than the pivot are on one side and elements greater are on the other. Then, we recursively apply the algorithm to the appropriate partition.

Algorithm: QuickSelect

1. **Choose a Pivot:** Select an element from **A** as a pivot. This can be any element, such as the first, last, or a random element.
2. **Partition the Array:** Rearrange the elements in **A** so that all elements less than or equal to the pivot come before the pivot, and all elements greater than the pivot come after it. Let j be the final position of the pivot after the partition.
3. **Divide and Conquer:**
    - If ***j*** equals **k - 1**, the pivot is identified as the **k**$^{\text{th}}$ largest element, and the algorithm returns this element.
    - If ***j*** is greater than **k - 1**, the algorithm recursively applies itself to the left partition, which consists of elements less than the pivot.
    - If ***j*** is less than ***k - 1***, the algorithm is recursively applied to the right partition, consisting of elements greater than the pivot, with an adjusted k to reflect the new position relative to the remaining elements.

Pseudo code

```
function quickSelect(A, k):
    if k > length of A:
        return nil
    return select(A, 0, length of A - 1, k)

function select(A, low, high, k):
    if low == high:
        return A[low]

    pivotIndex = partition(A, low, high)
    if k - 1 == pivotIndex:
        return A[pivotIndex]
    else if k - 1 < pivotIndex:
        return select(A, low, pivotIndex - 1, k)
    else:
        return select(A, pivotIndex + 1, high, k)

function partition(A, low, high):
    // Choose a pivot and rearrange elements
    pivot = A[high]
    i = low
    for j = low to high - 1:
        if A[j] <= pivot:
            swap A[i] and A[j]
            i = i + 1
    swap A[i] and A[high]
    return i
```

Example Run of QuickSelect

`A = [29, 28, 35, 20, 9, 33, 8, 9, 11, 6, 21, 28, 18, 36, 1], k = 6`

Initial Call: quickSelect(A, 6)

First Partition (Choosing 1 as pivot for simplicity):
Array after partition: `[1, 6, 9, 8, 9, 20, 28, 29, 11, 21, 33, 28, 18, 36, 35]`

Pivot position (1) is less than k - 1 (5). We need to look in the right partition.

Second Call: select(A, 2, 14, 6)

Second Partition (Choosing 9 as pivot):
Array after partition: `[1, 6, 8, 9, 9, 11, 18, 20, 21, 28, 28, 29, 33, 36, 35]`

Pivot position (5) is less than k - 1 (5). We need to look in the right partition.

Third Call: select(A, 6, 14, 6)

Third Partition (Choosing 18 as pivot):
Array after partition: `[1, 6, 8, 9, 9, 11, 18, 20, 21, 28, 28, 29, 33, 36, 35]`

Pivot position (6) equals k - 1 (5). We found the element: 18.